Xmorpion

- Rapport de projet -

Myriam Noël
Arnaud Grausem
Christophe Dessaint
Pierre-Emmanuel Pignède
&
Dimitri Dupuis-Latour

- Programmation en C -

Table des Matières

Préface: Bienvenue dans Xmorpion©
Énoncé - TP Noté #2 – Microprojet C

1) Présentation du travail

2) Partie Graphique

3) Partie Algorithmique

4) Intégration

Organisation du Travail

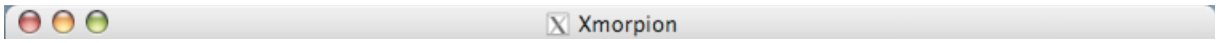
1) Mise en place du travail

2) Répartition des tâches

Explications du programme

1) Mode d'emploi du jeu

2) Descriptif des fonctions

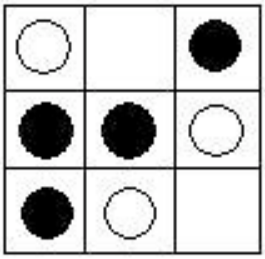


Préface: Bienvenue dans Xmorpion©

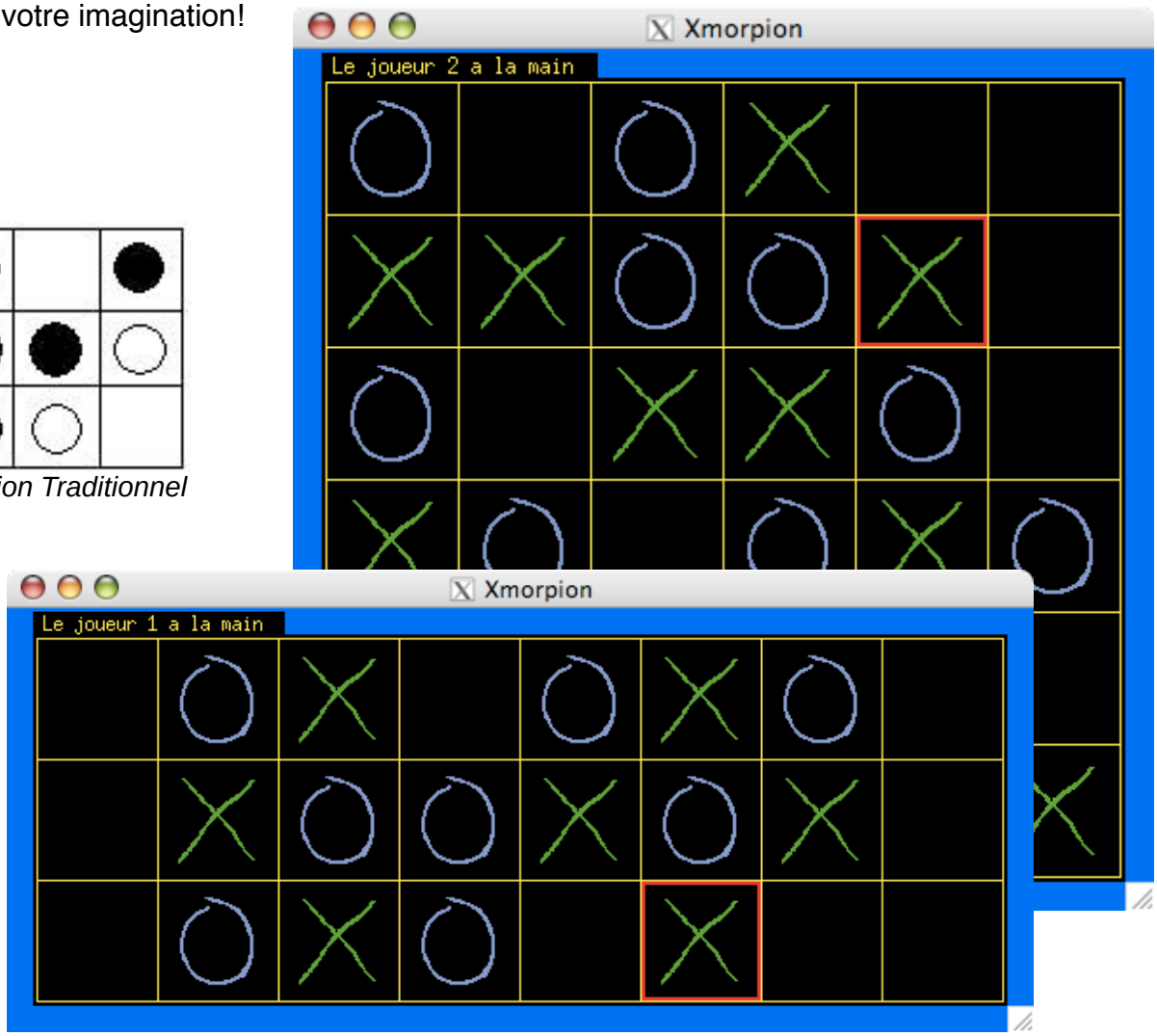
Vous venez d'acquérir Xmorpion©, et nous vous en félicitons ! Vous verrez à l'usage que c'est un très bon choix. Ce manuel utilisateur vous guidera tout au long de votre apprentissage de Xmorpion©.

Vue d'ensemble des fonctionnalités de Xmorpion:

Xmorpion permet de jouer au jeu du morpion seul contre l'ordinateur ou à deux. De plus, et contrairement au morpion traditionnel, il n'est pas limité à des tableaux de 3x3, ni même à des carrés d'ailleurs. On peut ainsi jouer sur des tableaux de 3x8, ou même 6x6. Laissez libre cours à votre imagination!



Morpion Traditionnel



Morpion évolué: Xmorpion© vous laisse jouer comme vous l'entendez !

Énoncé - TP Noté #2 – Microprojet C

❖ Organisation du travail

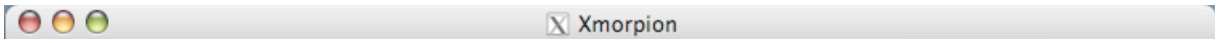
Vous réaliserez ce travail par équipes de quatre. Vous accompagnerez le code d'un texte décrivant vos structures et fonctions (sorte de manuel utilisateur de votre code) et me précisant "qui a fait quoi" : lors d'un travail en groupe, il est très important d'analyser le problème et de se rendre compte des difficultés de chaque question. La manière dont vous vous serez réparti le travail sera donc prise en compte.

En particulier, dans ce projet, il y a deux grandes parties : une partie algorithmique et une partie graphique. C'est en général une bonne idée de bien séparer les calculs de l'affichage. Cela correspond à deux styles de programmation très différents et, en étudiant un peu le problème, on se rend souvent compte que les deux parties sont décorrélées.

Je vous demanderai donc de faire deux équipes de deux : une équipe chargée de la partie algorithmique et une équipe chargée de la partie graphique. Normalement, des programmeurs expérimentés commencent par rédiger de manière commune l'interface entre les différentes parties d'un projet, à savoir dans ce cas, un fichier en-tête (.h) pour la partie algorithmique. Comme pour beaucoup d'entre vous, l'expérience est encore manquante, chaque équipe de deux va plutôt commencer par faire un programme séparé, puis vous passerez dans une deuxième étape à l'intégration en commençant par définir ce fichier d'en-tête.

Il va de soi que, comme d'habitude, la lisibilité de votre code ainsi que les commentaires que vous pourrez y faire, ou que vous écrirez dans le document joint, seront pris en compte dans la note finale.

Vous me rendrez le projet, comme d'habitude sous format électronique (document joint y compris). Prévenez-moi, avant de commencer à rédiger ce document, pour m'indiquer quel logiciel vous utiliserez pour ce faire afin de vous assurer que je pourrai le relire. Le projet est à rendre pour le ~~vendredi 25 février~~ (reporté au vendredi 4 mars 2005).



1) Présentation du projet

L'objectif de ce projet est d'écrire un jeu de morpion. Je rappelle que ce jeu se joue à deux, chaque adversaire jouant à tour de rôle pour cocher une case. Le plateau comporte trois lignes et trois colonnes de cases. Le jeu se remporte en alignant trois cases, verticalement, horizontalement ou diagonalement.

Votre programme offrira la possibilité de jouer à deux ou seul contre l'ordinateur. De plus, le jeu sera général dans le sens où vous donnerez la possibilité de jouer sur un plateau pouvant comporter plus de trois lignes et colonnes, et où le nombre de cases à aligner pour gagner pourra être de trois (comme dans la version classique) ou plus. Ceci vous permettra d'avoir une bonne base pour un jeu de puissance 4.

❖ 1.1 Les modules à écrire

Comme dit plus haut, il y a deux parties bien distinctes dans ce projet. Une première partie concerne l'affichage graphique et l'interaction. Elle conviendra aux personnes qui aiment la programmation d'interface et qui sauront s'adapter au style de programmation de tierces personnes (en l'occurrence, moi) car il faudra utiliser la bibliothèque d'outils graphiques que je mets à votre disposition. Cette bibliothèque ne comporte qu'un ensemble limité de fonctions et il faudra savoir jongler avec ces capacités limitées pour obtenir ce que vous désirez. Cette première partie concerne l'écriture du code principal (**`Xmorpion.c`**) ainsi que du module **`XGame.c`**

Une deuxième partie est plus algorithmique et concerne le jeu contre l'ordinateur. Elle conviendra aux personnes intéressées par la programmation d'algorithmes d'intelligence artificielle et désireux de faire leur code de A à Z : aucune bibliothèque ne sera mise à votre disposition, hormis la bibliothèque standard et il vous faudra écrire toutes les fonctions dont vous aurez besoin. Cette partie concernera l'écriture d'un programme principal de jeu de morpion en ASCII (afin de tester les choses) **`morpion.c`**, d'un module **`Game.c`**, ainsi que d'un module de gestion d'une liste chainée **`LIFOlist.c`** .

❖ 1.2 Outils mis à votre disposition

Vous baserez votre code sur l'embryon de programme que j'ai écrit pour vous : **`Xmorpion.c`**. Ce programme vous montre déjà comment ouvrir une fenêtre graphique dans laquelle on peut dessiner, et comment gérer les événements :

• "Exposition" : événement qui est généré automatiquement quand la fenêtre est, au moins, partiellement recouverte ou découverte. Il faut en général lancer une fonction qui redessine le contenu de la fenêtre. C'est le rôle de la fonction **`DrawIt`**. Dans le code que je vous donne, **`DrawIt`** redessine le tableau du jeu avec des cases cochées. A votre charge de changer le code par un appel à une fonction de dessin du jeu que vous écrirez dans le module **`XGame.c`**

• "Touche appuyée" : lorsque vous appuyez sur une touche alors que votre curseur de souris est dans la fenêtre, la fenêtre reçoit un événement clavier. Le code de **`Xmorpion.c`** appelle alors automatiquement la fonction **`KeyPressed`**. J'ai inséré un embryon de code vous permettant de voir comment réagir à diverses touches frappées (j'ai mis un **`case`** pour chaque touche que vous serez amenés à gérer dans le cadre du projet), en particulier, j'ai lié tout appui sur la touche "q" ou "Q" à la sortie du programme (appel à la fonction **`ExitApp`**). À vous de programmer les autres touches !

J'ai mis dans le code de **`Xmorpion.c`** les endroits où vous devriez insérer du code de manière à vous guider dans votre modification de ce fichier. Peu de modifications sont à réaliser. L'essentiel de votre travail consiste à écrire des modules qui seront utilisés par **`Xmorpion.c`**

J'ai encapsulé toutes les fonctions liées à l'interface graphique dont vous devriez avoir besoin. J'utilise volontairement la librairie Xwindow directement (même pas de Motif) afin que le code puisse être compilé sur tout système multifenêtrage basé Unix (qui utilisent tous Xwindow). Toutes les fonctions sont documentées dans **`Xresource.h`**

Il y a deux structures qui ne sont pas utilisées dans **`Xmorpion.c`** : les timers et les timeouts. Un timer est un objet qui est chargé d'appeler une fonction à intervalles réguliers jusqu'à ce qu'on l'arrête explicitement. Un timeout est un objet chargé d'appeler une fonction une et une seule fois après un intervalle de temps qui court à partir du lancement du timeout. Ces structures peuvent être utiles si vous voulez gérer des aspects temporels. A priori vous n'avez pas à les employer dans ce projet.

Enfin je vous donne également le Makefile vous permettant de taper juste **make** pour recompiler votre projet. Tapez **make clean** pour nettoyer les fichiers .o et le binaire.
À noter que ce Makefile fonctionne sous atlas... Si vous voulez compiler sous Linux, je vous fournis un Makefile pour Linux : tapez alors **make -f Makefile.linux**. Attention : il y a deux autres cibles

- **make -f Makefile.linux clean** : efface tous les fichiers .o (nettoyage)
- **make -f Makefile.linux depend** : génère automatiquement un fichier qui indique les dépendances des fichiers .c (quels .h y sont inclus...) Donc la première fois que vous compilez, ou lorsque vous ajoutez un nouveau module, sous Linux, il convient de taper **make -f Makefile.linux depend** avant de taper **make -f Makefile.linux**.Les sections qui suivent décrivent les deux parties indépendantes que vous aurez à programmer par sous-équipe. Une troisième section vous aide pour l'intégration. Prenez garde à tout lire avant de vous lancer dans la programmation car, même si les deux parties sont indépendantes dans un premier temps, il vous faudra garder toujours à l'esprit l'étape finale d'intégration. Pour la simplifier, il est donc sage de se mettre d'accord sur un minimum de structures : discutez donc de ce que vous mettrez dans vos structures principales avant de pianoter sur vos claviers...

A noter que la démarche ainsi que les algorithmes que je propose ne sont pas forcément optimaux. Comme dans tout projet, il vaut mieux débuter par quelque chose de simple même si sub-optimal mais qui marche, avant de se lancer dans les optimisations. Vous avez tout loisir de proposer des fonctions ou une architecture plus adéquates. Les seules contraintes que je mets sont :

- un code principal (fichier contenant la fonction main, c'est-à-dire Xmorpion.c et morpion.c) minimaliste. Il ne doit pas contenir de fonctions nouvelles et les fonctions seront complétées au minimum.
- une structure pour stocker les informations graphiques sur le jeu et gérer son affichage (appelée XGame ci-dessous).
- une structure pour stocker le jeu lui-même et les fonctions associées implantant les algorithmes d'intelligence artificielle (appelée Game ci-dessous).

Nous allons à présent entrer dans le vif du sujet : à vos claviers et vos neurones !

2) Partie graphique

Dans un premier temps, le but final de cette partie sera d'écrire un programme Xmorpion permettant à deux joueurs de s'affronter. La taille du tableau de jeu (en nombre de lignes et colonnes) sera configurable, ainsi que le nombre de cases à aligner.

❖ 2.1 Le module XGame

Le cœur de votre programme sera une structure nommée **`XGame`**. Cette structure et les fonctions associées permettant de la manipuler seront définies dans le module **`XGame`** formé des fichiers **`XGame.h`** pour l'en-tête et **`XGame.c`** pour le code.

La structure **`XGame`** doit contenir tout ce qui permet de décrire un jeu. Idéalement elle contiendra un pointeur vers une structure **`Game`** (voir la partie algorithmique) mais comme dans un premier temps les deux parties doivent être indépendantes, vous aurez à écrire un minimum de choses concernant la structure **`Game`**. Cette écriture peut être faite en commun entre les deux équipes et permettra de définir une base d'interface (au sens fichier .h) entre les deux parties.

Je vous laisse le soin de définir votre structure **`XGame`** et les fonctions associées. Je veux au minimum avoir les fonctions suivantes :

• **`XGame *newXGame(int nlines, int ncolumns, int nalign, Player p)`**; qui alloue une structure **`XGame`** pour un tableau de jeu de **`nlines`** lignes et **`ncolumns`** colonnes, que l'on gagne en alignant **`nalign`** cases. Le type **`Player`** sera à définir par vos soins et devra permettre d'indiquer quel joueur joue en premier.

• **`XGame *deleteXGame(XGame *xg)`**; qui libère la place mémoire allouée pour une structure **`XGame`** (voir la fonction précédente) et renvoie un pointeur vers **`NULL`**. • **`int setActiveXGame(XGame *xg, int line, int column)`**; qui rend active la case située sur la ligne **`line`** et la colonne **`column`**, c'est-à-dire que cette case sera cochée sur le joueur appuie sur la touche de retour chariot.

• **`int updateXGame(XGame *xg)`**; qui met à jour le jeu en cochant la case active. Le fait de savoir quel est le joueur courant sera géré de manière interne.

• **`int drawXGame(XGame *xg, Widget draw, int x, int y)`**; qui dessine le tableau du jeu dans la fenêtre X **`draw`**, le coin supérieur gauche du tableau de jeu ayant pour coordonnées (x,y) dans la fenêtre **`draw`**.

Cette dernière fonction est la seule qui mérite quelques explications car elle fera appel aux fonctions du module **`Xresource`** que je vous fournis. Il me faut commencer par définir ce qu'est une pixmap.

❖ 2.2 Les pixmaps

Une pixmap est tout simplement une image au sens de Xwindow. La fonction **`CreatePixmap`** dans **`Xresource.h`** permet de créer une pixmap vide de largeur et

hauteur données en pixels. Nous allons utiliser une pixmap vide pour le fond du tableau de jeu. Une pixmap a ceci d'intéressant que c'est un "drawable", c'est-à-dire un objet graphique X dans lequel on peut dessiner. Un autre exemple de drawable est la fenêtre (de type **Widget** et nommée **draw** dans **Xmorpion.c**) que vous passerez en paramètre de la fonction **drawXGame**. Dessiner dans un drawable veut dire y dessiner des lignes (fonction **DrawLines** de **Xresource.h** pour un Widget et **DrawLinesInPixmap** pour dessiner dans une pixmap) mais aussi y imprimer des pixmaps (fonction **PlacePixmapInWindow** pour imprimer dans une fenêtre **Widget** ou **PlacePixmapInPixmap** pour imprimer dans une pixmap).

Il existe un format très sympathique pour les petites images (icones) : le format XPM (fichiers d'extension .xpm). Il est sympathique parce qu'il code l'image en ASCII et qu'il la code dans une structure ... C ! Une image peut donc très simplement s'inclure au moyen de la directive de précompilation **#include**. Vous en avez un exemple dans **Xmorpion.c**. Le fichier **cross.xpm** contient l'icone codant une croix, utilisée pour cocher une case par un des joueurs (l'autre fait un rond, codé dans **circle.xpm**) : vous pouvez en visualiser le contenu par une simple commande **cat cross.xpm**.

Vous voyez que c'est simplement un tableau de chaines de caractères (comme argv !). La première chaine de caractères donne des informations sur l'image : largeur (en pixels, ici 64), hauteur (en pixels, ici 64), nombre de couleurs (2), nombre de plans (1). Les lignes suivantes font l'association entre un code ASCII (un caractère) et une couleur : nous avons donc un caractère, suivie de 'c' pour couleur, suivie du codage couleur dans ses trois composantes R(ed), G(reen) et B(lue), chacune codée en hexadécimal (précédée d'un #) entre 0 et 255, c'est à dire entre 00 et FF. La première couleur est donc associée au caractère ' ' (espace) et est la couleur transparente (NONE). La deuxième couleur est associée au caractère '.' et est en gros un rouge (R=FF=255, G=00, B=01). Ensuite, nous avons une chaine de caractères par ligne de l'icone. Chaque chaine de caractères contient donc autant de caractères qu'il y a de colonnes chacun de ces caractères codant la couleur d'un pixel.

La fonction **CreatePixmapFromData**, disponible par **Xresource.h**, permet de générer une pixmap à partir d'un tel fichier au format XPM : il suffit d'inclure ce fichier dans le programme C appelant **CreatePixmapFromData**; une nouvelle variable globale au module, et du nom de l'image est alors définie (pour l'icone ci-dessus, la variable est nommée cross_xpm) et il n'y a qu'à passer cette variable en paramètre de **CreatePixmapFromData** pour générer la pixmap appropriée. Voir **Xmorpion.c** pour un exemple d'une telle manipulation.

❖ 2.3 Travail à réaliser

Dans cette partie, vous aurez à compléter **`Xmorpion.c`** de manière à permettre le jeu entre deux joueurs. Pour ce faire, vous utiliserez une variable globale de type **`XGame`**. Les nombres de lignes, colonnes, nombre de cases à aligner, premier joueur seront passés en arguments du programme **`Xmorpion`**.

- La fonction **`DrawIt`** sera modifiée de manière à ne faire appel qu'à la fonction **`drawXGame`**. Les différents case de la fonction **`KeyPressed`** seront modifiés de manière à faire appel aux fonctions **`setActiveXGame`** (touches de flèches) et **`updateXGame`** (touche de retour chariot).
- La fonction **`main`** sera modifiée de manière à traiter les arguments du programme, allouer la variable globale de type **`XGame`** et créer la fenêtre de jeu.
- La fonction **`drawXGame`** dessinera dans la fenêtre passée en argument, une pixmap dans laquelle auront été dessinées des lignes pour délimiter les lignes et colonnes, puis affichera pour chaque case cochée la croix ou le rond (en fonction du joueur).Vous proposerez un mode d'affichage pour montrer quelle est la case active (celle que le joueur sélectionnera s'il tape sur retour chariot). Pour cela, vous pourrez par exemple définir de nouvelles pixmaps (case active vide, case active+croix, case active+rond...).

Les fonctions indiquées ci-dessus ne suffisent pas pour faire le jeu. A vous de définir et écrire les fonctions qui manquent ! (et n'oubliez pas de les documenter dans votre rapport final).

3) Partie algorithmique

L'objectif de cette partie est dans un premier temps d'écrire un programme qui permet de jouer au morpion contre l'ordinateur. L'affichage et l'indication des cases à cocher se fera uniquement sous forme d'entrée/sortie au format texte (affichage dans une fenêtre de commande par des **`printf`** et lecture par des **`scanf`**).

Le but plus profond est de définir une structure **`Game`** permettant de stocker et gérer le jeu, d'écrire l'algorithme d'IA permettant de déterminer le prochain coup de l'ordinateur, et d'écrire des petits modules associés, nécessaires au bon fonctionnement de tout ça. Le programme morpion vous offrira un cadre de validation de votre code.

Je commence par décrire l'algorithme d'IA (**`MinMax`**) que vous utiliserez, puis je décrirai le squelette de base de la structure **`Game`** que vous devrez écrire.

❖ 3.1 L'algorithme MinMax

J'ai déjà brièvement décrit cet algorithme en cours. La base du concept est que le jeu du morpion, tout comme le jeu de puissance 4 ou encore le jeu d'échec, peut se décrire sous forme de graphe :

- un nœud est une position
- les fils d'un nœud sont les positions autorisées pour le joueur suivant
- un nœud est terminal si la partie est gagnée par l'un des joueurs ou si le tableau est plein (toutes les cases sont cochées : match nul)

Une partie se code donc comme une suite de coups, chacun étant fils du précédent. L'algorithme **`MinMax`** permet de donner une valeur à chaque nœud du graphe, et ainsi, en comparant les valeurs de tous les nœuds fils d'une position donnée, déterminer quel est le meilleur coup à jouer. Cet algorithme n'est pas le plus performant, ni le plus efficace. Sa complexité est de plus assez importante, comme vous vous en apercevrez, mais son principe est simple et il fonctionne bien pour le morpion. Une fois que vous aurez codé cet algorithme, le codage futur (hors TP) d'algorithmes plus compliqués devrait être simplifié.

Le but de l'algorithme **`MinMax`** est donc d'attribuer une valeur à chaque nœud. Dans une partie à deux joueurs (Humain et Ordinateur), chaque nœud est soit H, soit O, selon que le dernier joueur à avoir coché une case est l'Humain ou l'Ordinateur. Tous les fils d'un nœud H sont O et vice-versa. On commence par attribuer à tout nœud terminal la valeur :

- $V(n) = -\infty$ pour les nœuds O gagnants
- $V(n) = +\infty$ pour les nœuds H gagnants
- $V(n) = 0$ pour les nœuds (H ou O) nuls. En pratique, l'infini étant difficile à coder, on utilisera les valeurs ± 1 (à modifier dans la version avec arbres tronqués).

On peut alors donner récursivement une valeur à chaque nœud en remontant dans l'arbre :

- pour un nœud H, $V(n) = \min_{p \in fils(n)} V(p)$
- pour un nœud O, $V(n) = \max_{p \in fils(n)} V(p)$

❖ 3.2 Implantation de MinMax

En pratique, l'arbre est virtuel, c'est-à-dire que vous n'allez pas commencer par calculer l'arbre entier du jeu et le garder en mémoire mais bien d'estimer pour chaque position sa valeur de manière récursive, l'arbre n'étant jamais stocké dans son entier en mémoire. Bien sûr ce n'est pas optimal, surtout que si on se débrouille bien, l'arbre n'est pas si gros que ça pour un petit jeu de morpion. Cependant, la taille de l'arbre grandit très vite. Pire, la complexité du calcul est exponentielle, et vous vous apercevrez que le calcul, rapide pour une taille de jeu 3 × 3 devient très lente pour un jeu 3 × 4. Il conviendra donc de modifier l'algorithme, ce que nous verrons plus tard. Enfin, nous n'avons pas vu les structures arborescentes, seulement les listes chainées, qui posent a priori déjà suffisamment de problèmes.

Étant donné une position courante, nous voulons donc calculer sa valeur. La formule est récursive par nature. La démarche que je vous propose est la suivante :

- obtenir la liste des positions accessibles
- pour chaque position accessible:
  - mettre à jour le jeu avec cette position (simulation de jeu)
  - calculer la valeur du nouveau jeu (appel récursif à **`MinMax`**)
  - défaire le jeu ("**`undo`**")
- calculer le max (ou min) sur toutes les valeurs renvoyées

Vous devrez donc être à même de stocker un ensemble de positions. De plus, vous devez avoir un mécanisme d'undo. Pour cela, la liste chaînée est votre amie !

En effet, une partie, à tout moment, peut se décrire comme une suite de coups, donc de positions jouées. Imaginons que vous ayez défini une liste chaînée de type **`LIFO`** (Last In First Out, voir dernier TD de C). Jouer un coup revient faire un **`push`** de la position indiquée dans la liste des coups. Défaire (**`undo`**) un coup, revient à faire un **`pop`** dans la liste des coups. De plus, cette structure peut également vous permettre de stocker la liste des coups accessibles à partir de la position courante.

❖ 3.3 Travail à réaliser

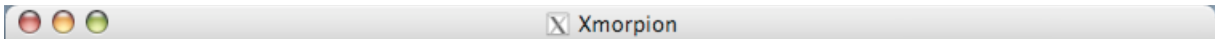
Vous commencerez donc par définir un module **LIFOList** permettant de gérer une liste chainée de positions. Un type **Position** pourra être défini afin de stocker un numéro de ligne et de colonne. Les fonctions à écrire pour cette liste seront les classiques **newList**, **deleteList**, **popList**, **pushList**. Vous écrirez ensuite le module **Game** qui définit la structure **Game**. Cette structure contiendra tout ce qui est nécessaire pour décrire un jeu : nombre de lignes, de colonnes, nombre de cases à aligner pour gagner, dernier joueur à avoir joué. Pour ce joueur, et de même que dans **XGame**, vous pourrez définir un type **Player**. Afin de décrire le jeu courant, vous utiliserez deux champs :

- un premier champ sous forme de tableau bidimensionnel, de la taille du tableau de jeu (en nombre de lignes et colonnes). Chaque case sera de type **Player**, en prenant garde à définir dans votre énumération un cas **None** à côté de **HUMAN** et **COMPUTER**. Le tableau étant petit, nous ne sommes pas contraints ici par des considérations de performance. Un tableau classique (pas forcément à lignes contiguës) est donc tout à fait adapté.
- une liste chaînée de positions qui garde en mémoire tous les coups joués

La fonction **updateGame** mettra à jour le jeu en cochant la case donnée en paramètre (liste chaînée, tableau de cases, et dernier joueur). La fonction **undoGame** fera de même mais à l'envers (aucune case n'est passée en paramètre dans ce cas : on obtient la position par un **popList** sur la liste des coups joués).

Vous aurez également besoin de définir des fonctions externes comme **isWin** et **isFull** pour savoir si le jeu est terminé (par un gain ou une partie nulle), ou encore **isAccessiblePosition** pour savoir si une position donnée peut être cochée dans le jeu.

Des fonctions internes devront être écrites pour le calcul **MinMax**, notamment un **nextMoves** qui renvoie la liste des positions accessibles à partir d'une situation de jeu donnée. Enfin, vous écrirez un programme **morpion**.c qui prend en paramètres les informations sur le jeu (nombre de lignes, de colonnes, de cases à aligner, premier joueur), crée une structure **Game** et permet de jouer contre l'ordinateur. Le jeu sera affiché sous format texte après chaque coup, un coup sera entré par l'humain par son numéro de ligne et de colonne.



4) Partie intégration

Cette partie est la partie finale où vous allez intégrer tous ce que vous avez fait afin d'avoir un programme graphique qui permet de jouer au morpion contre un joueur ou contre l'ordinateur. Deux choses seront à faire dans cette partie :

- modifier **`XGame`** pour intégrer la structure **`Game`** et épurer le module **`XGame`** de tous les aspects calculatoires déjà présents dans **`Game`**.
- améliorer l'algorithme **`MinMax`**. En effet, vous vous apercevrez que le temps de calcul devient prohibitif, même pour des petits tableaux de jeu (type 5 × 5). Une manière d'améliorer les choses est de faire appel à l'algorithme **`AlphaBeta`**, mais même celui-ci montre assez vite ses limites. La seule solution est de tronquer l'arbre de jeu (autrement dit la profondeur de récursion dans le calcul **`MinMax`**), ce qui revient à savoir évaluer une position de jeu non terminale. Vous modifierez la fonction **`MinMax`** pour qu'elle prenne en paramètre une profondeur de récursion maximale et vous proposerez une fonction permettant d'évaluer la valeur d'une position non terminale de jeu. Dans ce but, il faudra sûrement introduire des score intermédiaires entre les gains de l'un ou l'autre joueur. Vous pourrez donc utiliser un gros score, de type ±100000 ce qui vous laissera de la place pour qualifier les situations intermédiaires.

Pour la rédaction de ces parties, il sera sûrement bénéfique d'inverser les équipes : l'équipe qui aura fait la première partie graphique se chargera d'améliorer l'algorithme **`MinMax`** et l'autre équipe, chargée originellement de la partie algorithmique, effectuera l'intégration.

Organisation du Travail

❖ 1) Mise en place du travail

Formation des deux groupes. Dès que l'équipe fût formée, on a rapidement pu former les deux équipes Algo et graphismes dans la mesures où certains étaient très tentés par l'une ou l'autre des spécialités.
Comme il était conseillé, chacun a d'abord lu et relu plusieurs fois l'énoncé pour bien le comprendre (notamment les fonctions de Xresource) et bien s'en imprégner. Une fois que l'on avait saisi le fonctionnement du jeu on a défini les fonctions, communes aux deux groupes, dont on aurait besoin. Les prototypes de ces fonctions (isFull, isWin…), ainsi que la structures Game, furent définies ensemble, puis regroupés dans le fichier d'en-tête Game.h, fichier qui servit aux deux groupes avant l'intégration finale.

Difficultés rencontrées. Le développement fut rendu difficile par le fait que l'on a pu faire fonctionner X11 que sur un seul des ordinateurs du groupe, et que donc les compilations intermédiaires et finales durent se faire uniquement sur celui-ci. De plus les salles-machines furent fermées juste avant la date de remise du TP à cause de vols qui s'y seraient produits. De ce fait l'intégration n'a pas pu être poussée à son terme: chacunes de leur côté, la partie algo et la partie graphique compilent et s'exécutent (en mode texte pour la partie algo). Mais lorsqu'il a fallu intégrer les deux, seuls les tests de fin de partie (isWin et isFull) purent être intégrés. Cela permet néanmoins de jouer graphiquement une partie à deux joueurs sans problèmes.

Notions abordées. Malgré cela, nous sommes assez content du résultat, dans la mesure où ce TP nous a permis (enfin) de créer un programme **graphique** (ça change des programmes en mode texte). De plus, on a pu découvrir ou approfondir certaines notions:
- la librairie X11
- la programation événementielle (avec les EventsHandler))
- la programmation modulaire (bien rédiger les headers, les #ifndef, make et les makefile…)
- l'algorithme MinMax, qui d'après la littérature circulant sur internet est un algorithme très connu.
- … et permis aussi de (re)voir en détail certaines notions du langage C

❖ 2) Répartition des tâches

La répartition des taches fut assez aisée, en fonction des goûts de chacun.
Partie Graphique:
- Dimitri Dupuis-Latour (XGame, Xmorpion, Rapport PDF)
- Myriam Noël (Menu.c et .h pour initialiser le jeu)

Partie Algorithmique:
- Christophe Desaint (LIFOList)
- Arnaud Grausem (MinMax(), is Full(), isWin())
- Pierre-Emmanuel Pignède

❖ Known issues

Faisons le point sur ce qui fonctionne et ne fonctionne pas (encore):

Passage des arguments. Au lancement du programme, on peut mettre un nombre incorrect d'arguments, des arguments avec des lettres (au lieu d'entiers), des arguments contradictoires (on joue seul mais on veut être le joueur 2, on veut aligner plus de ligne qu'il n'y a de cases …)), et tout est géré. Donc c'est OK

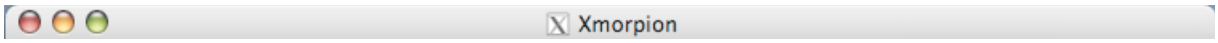
Tableaux carrés d'un nombre quelconque de cases. Ils sont crées normalement, les conditions de victoires ou de match nul y sont reconnues tout le temps. Donc OK

Tableaux pas carrés. Ils sont crées normalement, et on s'y déplace sans problèmes. Le match nul est détecté sans problème. Mais le programme ne détecte pas toujours les victoires en diagonales et anti-diagonales, lorsque les diagonales gagnantes ont une case sur la première ligne ou la première colonne, à l'exception de la case (0,0). Le problème vient apparemment d'une erreur d'indice dans la fonction `isWin()`, mais on n'a pas pu la trouver à ce jour.

Gestion des événements. La touche q permet toujours de quitter le jeu, et l'ordre d'apparition des écrans (écran de bienvenue, plateau de jeu, résultats -avec désactivation du clavier-, écran de fin et sortie de l'application sont maintenant gérés sans problème.)

Les fonctions de l'IA. Celles-ci fonctionnent en mode texte (avec `morpion.c`). Cependant le module `Game.c` actuel est un mélange des deux `Game.c` précédants, ce qui ne lui assure plus de compatibilité avec morpion.c.

Jouabilité. Xmorpion inclut certaines fonctions de la partie algorithmique (`isWin()` et `isFull()`), mais pas son IA. On ne peut donc pas jouer "contre l'ordi" (ce paramètre est néanmoins accepté par la ligne de commande.). Cependant une partie à deux est tout à fait possible, et la victoire ou le match nul sera détecté, géré et affiché.



Explications du programme

❖ 1) Mode d'emploi du jeu

Démarrer le jeu:
Si le programme est compilé[1], lancez dans le terminal la commande:
`Xmorpion <lignes> <colonnes> <nalign> <config> <Player>`
Xmorpion prend les paramètres suivants :

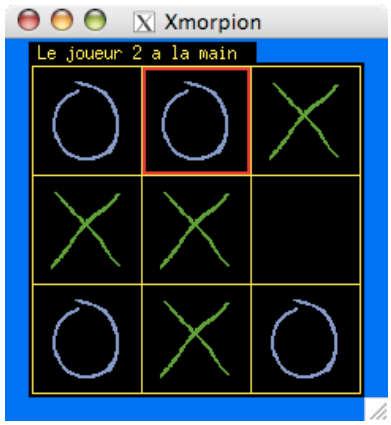
- **`<lignes>`**: nombre de lignes
- **`<colonnes>`**: nombre de colonnes
- **`<nalign>`**: nombre de cases à aligner (nécessairement plus petit ou égal à **`lignes`** et **`colonnes`**)
- **`<config>`**: c'est la configuration, ou aussi le nombre de joueur. on Joue à 1 ou à 2
- **`<Player>`**: c'est le joueur qui commence. S'il n'y a qu'un joueur, c'est nécessairement le premier joueur.

Écran de bienvenue:
Si les paramètres sont valides, vous aurez cet alors cet écran. Il est utile, et pas uniquement pour faire joli. Il permet de familiariser l'utilisateur avec deux concepts touchant à l'interface de ce jeu:

- Il faut placer le pointeur de la souris dans la fenêtre pour que celle-ci réponde;
- On l'invite à utiliser les flèches pour qu'il comprennent que c'est par cet interface qu'il communique avec le jeu et pas avec la souris.

Il faut donc presser une flèche…

Plateau de jeu
La fenêtre est alors recadrée à la dimension de votre plateau de jeu.
Le joueur qui commence est indiqué en haut. Tout au long du jeu le joueur qui à la main sera indiqué à cet endroit. On y verra aussi des messages d'erreur (case occupée) ou le message de fin de match.
Les flèches vous permettent de déplacer la case active, symbolisée par une bordure rouge. Une fois votre case choisie, pressez Enter. c'est au tour de l'autre joueur maintenant, sauf si votre case n'était pas libre.



Bon jeu…

---

[1] Pour compiler le programme sous Atlas ou Linux, cf à la fin, page…

❖ 2) Descriptif des fonctions

Partie Algorithmique

❖ Morpion.c et Morpion.h

Morpion.c est un programme qui prend en paramètres les informations sur le jeu :

- nombre de lignes (nlines)
- nombre de colonnes (ncolumns)
- nombre de cases à aligner (nalign)
- premier joueur (player)

Ces paramètres sont initialisés dans la ligne de commande, par la commande suivante :

`Syntaxe: morpion <lignes> <colonnes> <nombre_de_pions_à_aligner> <Player>`

Une fois les paramètres rentrés, le programme crée une structure Game qui permet de jouer contre l'ordinateur.

On entre le coup de l'humain grâce à son numéro de ligne et de colonne.

Ce programme sert principalement à tester les fonctions algorithmiques.
Il n'est pas indispensable au bon déroulement du jeu (mais conseillé).

Les fonctions implantées dans ce programme sont :

- **`void message_erreur()`** : affiche un message d'erreur en cas de case non jouable
- **`int message_gagne()`** : affiche un message en cas de victoire
- **`int avert_gain()`** : teste si l'alignement à réaliser pour la victoire est plus important que la dimension du tableau
- **`int saisi_position(int a, int b)`** : Permet de gérer l'interfaçage de récupération de coup de l'homme
- **`int Ordi(int a, int b)`** : fonction qui fait jouer l'ordinateur en utilisant minimax (qui utilise les coups possibles)
- **`int ecrit_coup_cpu()`** : écrit le coup joué par l'ordinateur dans le tableau de jeu
- **`int deroulement()`** : fonction qui gère l'alternance des joueurs

❖ Game.c et Game.h

Game.c est un module qui permet de gérer la partie algorithmique du jeu. Il implémente plusieurs fonctions et structure nécessaires au bon déroulement du jeu :

- **`struct Game`** : crée une structure permettant de connaître les informations sur le jeu à tout moment du jeu. En effet, elle permet de connaître le dernier joueur à avoir joué, de savoir quelle case est occupée et par qui, grâce au tableau d'entier ainsi que toutes les informations nécessaires au jeu (nombre de colonne, de ligne et de cases à aligner du morpion).
- **`extern Game *newGame(int nlines, int ncolumns, int nalign, int p)`** : alloue une structure Game pour un tableau de jeu de nlines lignes et ncolumnes colonnes, que l'on gagne en alignant nalign cases. Le type Player sera à définir par nos soins et devra permettre d'indiquer quel joueur joue en premier
- **`extern Game *deleteGame(Game *g)`** : libère la place mémoire allouée pour une structure Game et renvoie un pointeur vers NULL.
- **`int ** alloueTab(int nl, int nc)`** : met à jour le jeu en cochant la case active. Le fait de savoir quel est le joueur courant sera gérée de manière interne.
- **`int isWin(Game *g, int joueur)`** : fonction caractérisant les conditions de victoire
- **`int isFull(Game *g)`** : fonction caractérisant le cas du match nul
- **`int isAccessiblePosition(Game *g, int i, int j)`** : fonction permettant de savoir quelle case est libre
- **`void updateGame(Game *g, int i, int j, int joueur)`** : met à jour le jeu avec le joueur ayant joué et la case qui a été jouée et stocke la position jouée dans la pile
- **`void undoGame()`** : dejoue un coup
- **`void nextMoves(Game *g)`** : stocke dans la pile les coups accessibles
- **`int minimax(Game *g, int joueur, int *mi, int *mj)`** : fonction générant l'IA du jeu

❖ LIFOList.c et LIFOList.h

LIFOList.c est un module implémentant des fonctions gérant une pile LIFO.

- **`struct Position`** : crée une structure position dont les éléments sont un entier ligne, un entier colonne et un pointeur vers une position.
- **`struct LIFO`** : crée une structure lifo dont les éléments sont un pointeur vers la position de tête de la liste
- **`extern lifo *newList(lifo *liste)`** : alloue une structure liste
- **`extern position popList(lifo *liste, position *p)`** : dépile une position de la liste et renvoie la position dépilée
- **`extern void deleteList(lifo *liste)`** : libère la place mémoire allouée pour une structure lifo
- **`extern void pushList(int i, int j, lifo *liste)`** : empile une position sur une liste donnée

Partie Graphique

❖ Quelques mots sur X11…

Le système de fenêtrage X (X Window System), plus souvent appelé X11, est quasiment le seul système de fenêtrage utilisé par toutes les applications graphique sur tout système basé UNIX.

Mac OS X, basé UNIX, utilise cependant Quartz comme “Window Manager”, mais il peut également exécuter simultanément des programmes X11 grâce aux librairies X11 et à un serveur X basé sur le projet OpenSource XFree86. Nous le précisons car c’est sur cette plateforme qu’a été développée l’interface graphique de Xmorpion©. (et ce beau PDF ;¬))

Toute les fonctions d’affichage des graphismes de Xmorpion© utilisent les librairies X11, le rendant ainsi portable sur de nombreuses plateformes (Mac OS X, Linux, SunOS (Atlas), et toutes plateforme UNIX implémentant X11.) Malgré tout, cela ne fait que 5% des ordinateurs…

❖ Xmorpion.c et Xmorpion.h

Le module **`Xmorpion`** est le code principal, c’est lui qui contient la fonction **`main()`** qui est appelée au lancement du programme. Il permet à deux joueurs de jouer l’un contre l’autre. Il contient aussi deux autres fonction, **`DrawIt`** et **`KeyPressed`**, qui sont appelées par les “events handlers”, et qui elles-mêmes appellent respectivement **`draxXGame`** et **`setActiveXGame`**.

Main: traitement des arguments passées en ligne de commande, création des fenêtres, lancement du jeu, création des **`eventsHandler`**.

DrawIt: affiche le plateau de jeu, les messages, les écrans d’entrée et de sortie, et redimensionne les fenêtre quand il le faut. Beaucoup de ces actions sont déléguées à **`drawXGame`**.

KeyPressed: déplace la case active et vérifie qu’elle reste dans les limites du tableau. Dès qu’il y a victoire ou match nul, le clavier est “bloqué”, c’est à dire que seul la touche Q est activable. Sans cela, le jeu continuait et l’affichage devenait chaotique car ça n’a pas de sens de continuer le jeu alors que la partie est finie.

❖ Xresource.c et Xresource.h

Le module **`Xresource`** (.c et .h) définit les fonctions d'affichage graphique utilisées par le programme; ces fonctions sont basées sur la librairie X11. Ces deux fichiers n’ont pas été modifiés; de plus ni les **`timers`** ni les **`timeout`** n’ont étés utilisés (Même si on aurait pu).

❖ XGame.c et XGame.h

Le module **XGame** stocke les informations graphiques du jeu (dans sa structure **XGame**) et gère son affichage. La structure XGame contient:

- Game *g; //pointeur sur une structure Game, qui gère tout ce qui n'est pas graphique.
- XPoint caseActive;
- Pixmap bkg, cross, circle, border, credits; //différentes images, dont l'écran d'éntrée.
- int bkgwidth, bkgheight;
- int case_width, case_height;
- int border_width, border_height;
- int borderwidth;
- int debut; //marque le début du jeu: vaut 0 tant que l'on est sur l'écran d'intro, 1 après.
- int fin; //marque la fin du jeu: fin de la partie(=1), ecran de sortie(=2), sortie de l'application(=3)
- int config; //1 pour un joueur contre l'ordi, 2 pour 2 joueurs face à face
- char *msg; // le message à afficher, comme "le joueur 1 a la main" ou "Match nul !"

Le module XGame contient lui:

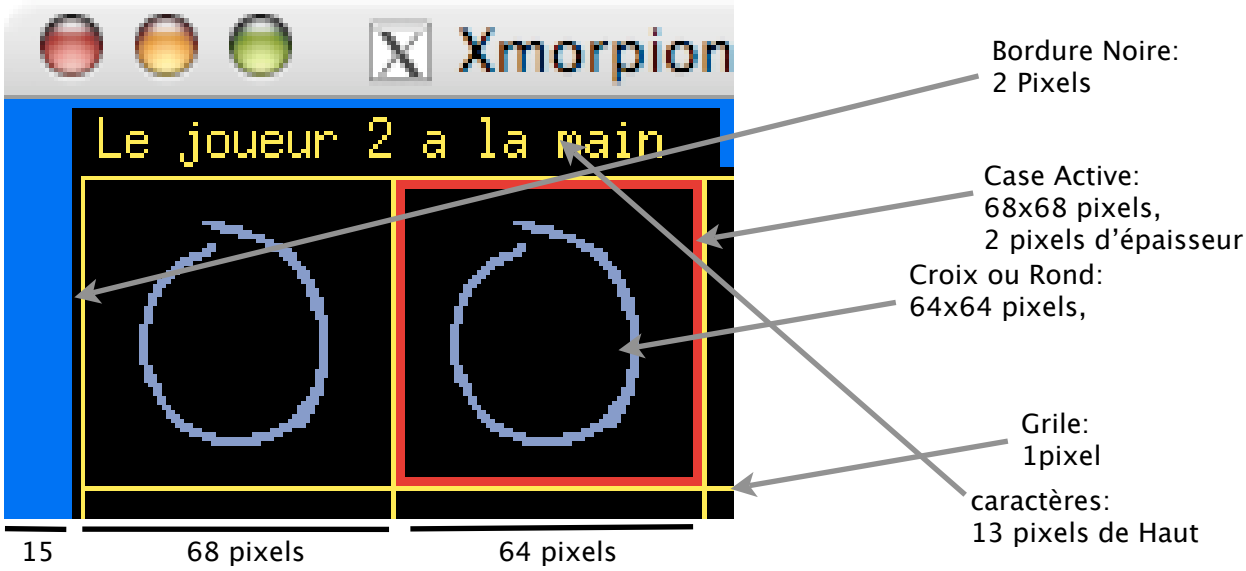
```
extern XGame *newXGame(int nlines, int ncolumns, int nalign, int p,
int config);//alloue une structure XGame

extern XGame *deleteXGame(XGame *xg);
//libère la place mémoire allouée pour une structure XGame et
renvoie un pointeur vers NULL.*/

extern int updateXGame(XGame *xg); //coche la case active.

extern int setActiveXGame(XGame *x); //rend active la case située
sur la ligne line et la colonne column,c'est-à-dire que cette case
sera cochée sur le joueur appuie sur la touche de retour chariot.

extern int drawXGame(XGame *xg, Widget draw, int x, int y);//dessine
le tableau du jeu dans la fenêtre X draw,
```

❖ Game.c et Game.h

Le module Game stocke (dans sa structure Game) toutes les les informations du jeu autres que graphiques et gère le bon déroulement du jeu. Il a déjà été décrit dans la partie algorithme.

❖ Remarques sur l'affichage

La fonction drawXGame est parsemée de constantes numériques telles que: `xg->borderwidth+5+j*(xg->case_width+5)`. Pour essayer d'expliquer pourquoi il y a des + 1, +5, +13 disséminés un peu partout sans raison apparentes, voici un schéma qui montre comment est construite mon interface.



❖ Les écrans de bienvenue

À la place des écrans de bienvenue et de sortie, il était prévu de faire au départ une animation en introduction. Nous y étions presque, mais devant le retard que prenait le reste du projet, on a laissé de côté, estimant que la partie graphique était déjà suffisamment au point.

Comment afficher un petit film avec X11 ? Voici comment on a procédé:

- Avec un logiciel de présentation (Keynote, le PowerPoint d'Apple) on a fait une animation avec des titres qui entrent et qui sortent. On l'a exporté en .mov (disponible en PJ peut-être).
- Avec QuickTime, on a convertit ce film en une suite d'images .png.
- Avec un logiciel de traitement d'images (Graphic Converter), on a d'abord réduit le poids des images (descente à 256 couleurs, réduction du nombre de pixels), puis on a crée une tâche pour automatiser la conversion en .XPM.
- Au final on se retrouve avec 43 images .XPM; on comptait afficher rapidement à l'aide de timers plusieurs images par seconde, et créant ainsi l'illusion du mouvement.

Peut-être pour une version 22 ?

---

[2] On va essayer de joindre ces fichiers et/ou le .mov avec le rapport, si la taille des pièces jointes n'est pas trop limitée. En tous cas ils existent !

Autres fichiers

❖ Makefile

Le fichier Makefile sert à compiler les sources en un exécutable UNIX. Ce fichier est celui recherché lorsque l'on invoque la commande make. On dispose de 3 Makefile:

- Makefile.linux : pour compiler sous Linux
- Makefile.sun : pour compiler sous Sun (machine atlas)
- Makefile : qui est une copie de Makefile.sun

- Pour compiler, se placer dans le répertoire où est le Makefile, et faire successivement:

gmake depend
(cela crée un fichier depend.inc qui définit automatiquement les dépendances entre les fichiers .o et les fichiers .c, .h et .xpm. En effet, une règle de Makefile du style .c.o: ne définit une dépendance qu'entre le .o et le .c et pas les .h)

gmake
(cela compile tous les fichiers objets (.o) et fait l'édition de liens pour générer le binaire, uniquement si c'est nécessaire)

gmake clean
(si vous souhaitez faire le ménage en effaçant tous les .o et le fichier depend.inc.)

- Pour compiler sous Atlas, il faut ajouter à son .login la ligne:

setenv LD_LIBRARY_PATH /usr/local/dp/gnu/gettext/lib:/lib:/usr/lib:/usr/local/lib:/usr/local/X11R6/lib
(Rajoute le répertoire : /usr/local/dp/gnu/gettext/lib dans la variable d'environnement $LD_LIBRARY_PATH)

- Pour compiler sous Linux, faire une copie de Makefile.linux en Makefile. Cela évite d'avoir à taper gmake -f Makefile.linux à chaque fois que vous voulez recompiler (par défaut, gmake prend le fichier Makefile...)